PPOOGL

Florent de Dinechin

Les subtilités de l'OO
(ou : la guerre des langages)

Introduction

Open issues in OO languages

Coupé de l'intro au cours *Advanced issues in object oriented programming* du Collège Impérial (cf web) :

- class based vs object based languages
- static types vs dynamic types
- structural types vs name types
- static binding vs dynamic binding
- inheritance vs delegation
- polymorphism through inheritance
- polymorphism through parameterised abstract data types
- subtypes vs subclasses, and interface inheritance vs implementation inheritance
- information hiding and alias protection
- multiple inheritance
- separate compilation, static linking vs dynamic linking, open world vs closed world
- object reclassification, roles, and self-modifying code
- combination with other paradigms

Most of above not exclusive to oo paradigm.

Avertissement

On peut très bien construire tout un projet sans avoir le moins du monde à se frotter aux horreurs ci-après décrites.

C'est même encouragé.

Je veux vous donner une idée de ce qui alimente la guerre des langages 00, vous n'êtes pas obligés d'y prendre part.

Typage et liaison statique ou dynamique

Statique ou dynamique

- Statique : connu à la compilation
- Dynamique : connu à l'exécution

(attention, on utilise les mêmes mots pour un autre concept : membre *de la classe* vs membre *de l'instance*).

Commençons par le typage

```
Philosopher extends Person (...)
Person somebody ;
somebody = new Person ;
somebody ; // static type Person, dynamic type Person
if(...) somebody = new Philosopher ;
somebody ; // static type Person, dynamic type Philosopher
```

Les choses se gâtent dès que Person/Philosopher ont des méthodes.

Eiffel, langage OO pionnier, essuie les plâtres

```
class Person { Book like; }
class Philosopher extends Person { PhilBook like; }
class Book{ }
class PhilBook extends Book { void print(){...} }
```

- En Eiffel, le `like` de la classe `Philosopher` *surcharge (overrides)* le `like` de la classe `Person`. OK ?
- Donc le `like` des `Philosopher` est de type `PhilBook` alors que celui des `Person` est de type `Book`. Pourquoi pas ?

Mézalors

```
Person somebody ;
Philosopher plato ;
plato=new Philosopher ;
plato.like=new PhilBook ;
if (...) somebody=plato ; // copie de pointeur
somebody.like=new Book ;
plato.like.print() ; // ERREUR ! !
```

C'est très mal pour la raison suivante :
Cela montre que le système de type d'Eiffel n'est pas *sain (sound)* :
Un système de type est sain si le type dynamique T' d'un terme déclaré de type T est toujours un sous-type de T
(autrement dit on a toujours que l'objet à l'exécution *est-un* T)

Sinon vous vous convaincrez facilement que cela casse toute mes beaux discours sur l'encapsulation.

Remarques anecdotiques

- `somebody=plato` peut (doit) vous sembler louche, mais c'est indispensable pour les boucles d'affichage de mon straterisk.
- Les dangereux théoriciens pénibles intégristes de la pureté vexatoire des types avaient prévu le truc en 1985.
- Eiffel fut conçu avec ce problème en 88.
- Le problème fut exposé en 90.
- Eiffel eut bien du mal à offrir une solution préservant la compatibilité ascendante.
- On a trouvé aussi quelques failles dans la sécurité de Java pour des raisons similaires (mais au niveau de la machine virtuelle).

Moralité : Il faut toujours écouter les théoriciens.

Solutions à ce problème (1)

Les membres de la surclasse ne peuvent pas être redéfinis par la sous-classe.

Solution psychorigide...

Solutions à ce problème (2)

Les champs de la surclasse sont cachés mais pas surchargés par ceux de la sous-classe :

- on objet `Philosopher` a deux champs `like` :
  - un de type `PhilBook`,
  - et un de type `Book` qui est celui utilisé lorsque le `Philosopher` est considéré comme une `Person` comme les autres.
- un objet de la sous-classe peut accéder aux deux
- et accède à celui donné par le type statique.
- C'est en gros ce que font C++ et Java.

```
if(...) somebody=plato ;
somebody.like // static type : Book
plato.like // static type : PhilBook
```

Solutions à ce problème (3)

Un champ peut être redéfini dans la sous-classe, *uniquement avec un sous-type* du champ correspondant de la surclasse, et dans ce cas seulement on surcharge. Seulement, dans une affectation, il faut vérifier *à l'exécution* le type de gauche et le type de droite.

```
//PhilBook est bien un sous-type de Book
somebody=new Person ;
somebody.like = new Book // OK
if(..) somebody=plato ; // copie de pointeur
somebody.like = new Book // dynamic error
```

(pas très compositionnel, tout cela)

Les tableaux Java (qui, pour de sordides raisons d'efficacité, ne sont pas des classes mais des constructeurs de types) font quelquechose qui ressemble à cela. Je vous rassure, cela peut très bien vous passer au dessus de la tête.

Solutions à ce problème (4)

Yen a encore deux autres, dont celle (bâtarde) d'Eiffel dans les transparents de Sophia Drossopoulou au Collège Impérial.

Ah, et il y a aussi la solution de Python : aucun typage statique, et démerde-toi pour débugger à l'exécution.

Amusant : Sophia Drossopoulou dit de C++

"*type system is (probably) sound.*"

Et maintenant, liaison statique ou dynamique ?

Liaison *(binding)* c'est la liaison des méthodes aux objets

Liaison statique ou liaison dynamique

- Liaison statique : on appelle les méthodes de la classe déclarée de l'objet.
  - plus efficace car le code à appeler est connu à la compilation
  - par défaut dans C++
- Liaison dynamique : on appelle toujours les méthodes de l'objet
  - le plus proche de la philosophie orienté-objet
  - conceptuellement plus simple
  - surcoût d'implémentation : rajouter à chaque objet un pointeur (caché) qui pointe vers une table des méthodes de la classe de l'objet.
  - C'est ce que font Java et Python
  - C'est ce que fait C++ pour les méthodes déclarées *virtuelles*

Pour C++, la philosophie du C est respectée : on n'aura le surcoût que si on en a vraiment besoin, et on l'aura effectivement exprimé.

Motivations de C++

(d'après Bjarne Stroustrup)

- facile à comprendre pour les gens qui connaissent C
  - On a ainsi conservé toutes les déficiences du C.
- facile à implémenter pour les gens qui font des compilateurs *(should be implementable without using an algorithm more complicated than linear search)*
  - Considérant gcc, c'est un échec.
  - L'inconvénient, comme le montre la suite, c'est que cela oblige le programmeur, lui, à faire des parcours de graphes compliqués dans sa tête.
  - Du reste, je n'ai pas remarqué que les gens qui faisaient des compilateurs étaient effrayés par les algorithmes compliqués. Au contraire, ils adorent cela.

Héritage en C++

- Tout le typage est vérifié à la compilation (statique) pour la performance
- Comme c'est du C, il y a des pointeurs (je passe sur les références)
- Assignation = copie champ à champ (on peut le surcharger pour que la copie suive les pointeurs)
- Assignation de pointeur = copie d'adresse.
- Appel de méthode d'un objet `a.f()` : la fonction est connue à la compilation.
- Appel de méthode d'un objet connu par un pointeur `ap->f()` :
  - statique (d'après le type de `ap`) par défaut
  - dynamique (d'après le type de `ap*`) si `f()` est déclarée *virtuelle* (attention, ce n'est pas le *virtuel* que l'on a vu jusque là)

Les méthodes virtuelles en C++

```
class A{
 public:
  virtual int f( ) {return 100;};
  int g( )         {return 150;};
};

class B: public A{
 public:
  virtual int f( ) {return 200;};
  int g( )         {return 250;};
};

class C: public B{
 public:
  virtual int f( ) {return 300;};
  int g( )         {return 350;};
};
```

```
void main(){
  // static binding for (value) objects
  B b1;
  A a1;
  a1.f(); // returns 100
  a1.g(); // returns 150
  b1.f(); // returns 200
  b1.g(); // returns 250
  a1 = b1;
  a1.f(); // returns 100
  a1.g(); // returns 150
}
```

Les méthodes virtuelles en C++, suite

```
class A{
 public:
  virtual int f( ) {return 100;};
  int g( )         {return 150;};
};

class B: public A{
 public:
  virtual int f( ) {return 200;};
  int g( )         {return 250;};
};

class C: public B{
 public:
  virtual int f( ) {return 300;};
  int g( )         {return 350;};
};
```

```
void main(){
  // dynamic binding for pointers
  B* bp = new B();
  A* ap1 = new A();
  A* ap2 = new C();
  ap1->f(); // returns 100
  ap1->g(); // returns 150
  ap2->f(); // returns 300
  ap2->g(); // returns 150
  bp->f();  // returns 200
  bp->g();  // returns 250
  ap1 = bp;
  ap1->f(); // returns 200
  ap1->g(); // returns 150}
```

On peut continuer longtemps

`virtual int f( ) { return g() ;} ;`
C'est le `g()` de `A` ou le `g()` de `B` ?

Je vous laisse lire la doc si cela vous amuse...

Et je vous invite à éviter ce genre de situation.
Voire à éviter C++.

Héritage en Java

- Tout le typage est vérifié à la compilation (statique) pour la performance
- Tous les objets sont des pointeurs
- Assignation = copie de pointeur
- Appel de méthode d'un objet `a.f()` : liaison dynamique

Moi je préfère.

Remettons en une couche

Héritage multiple : “Une orange est-un fruit *et* est-une sphère”

- Sujet de beaucoup de baston à l'époque de la conception de C++
- Genre de problème : le losange de la mort

```
class Top{
 virtual void f(){ cout << "Top::f()"  ;}
};

class Left : public virtual Top{
 void g(){ f();}
};

class Right : public virtual Top{
 virtual void f(){ cout <<  "Right::g()"  ;}
};

class Bottom : public Left, public Right{ };

(...)

Bottom *b; b->g();
```

Ptites stouquettes

- Java s'en sort par des *interfaces* :
  - classes complètement abstraites (même pas d'attributs, sauf si constants)
  - On peut hériter de plusieurs interfaces, et on doit alors implémenter toutes les méthodes de chaque. Comme on n'hérite d'aucune *implémentation* de méthode, tous les problèmes disparaissent.
  - Pour l'orange, on abstrait le fruit, la sphère, ou les deux ?
- Python s'en sort par "la première implémentation qu'on rencontre par un parcours des ancêtres de gauche à droite, en profondeur d'abord"
  - Exercice : comparez les fonctionnements du diamant et de son image miroir
  - Le "en profondeur d'abord", c'est pas ce qu'on voudrait, c'est juste plus facile
- Pour C++ je vous renvoie à la FAQ Lite

Mon modeste point de vue

- Intéressants problèmes théoriques, surtout pour la sécurité
- En pratique, si vous avez un héritage multiple de deux méthodes qui ont le même nom, c'est sans doute que quelqu'un a été paresseux dans le choix des noms des méthodes.
  - Plutôt que de m'extasier sur votre maîtrise du diamant de la mort, je saquerai la paresse :)
  - Variante OO de la guerre contre les programmes où toutes les variables s'appellent `tmp1` ... `tmp3224` et toutes les fonctions s'appellent `f` ou `g` ...

Conclusion

Moralité de tout cela

- Ne croyez pas que j'ai soulevé tous les problèmes/questions.
- Une approche modeste de votre projet vous évitera tous ces tracas (mais je suis certain que cela vous a plu).
- Autrement dit : faudrait pas remplacer les bugs que vous économisez en faisant de l'OO par des bugs que vous rajoutez en faisant des héritages et des surcharges capilitractés.
- Ce sont des problèmes importants pour la *sécurité* : Java est en principe conçu pour qu'une applet Java ne puisse pas lire mon disque dur, même si un vilain hackeur fait des cochonneries auxquelles personne n'avait pensé tellement elles paraissent stupides.
- Les preuves de (bonnes) propriétés passent par des hirschkofferies à coucher dehors, ce n'est pas l'objet de ce cours (ouf).
- On n'a pas fini de concevoir des langages de programmation
- Java ou Python pour ce projet, pas C++ (je l'ai déjà dit ?)

Bonus : La FAQLite de C++ fait rien qu'à me contredire

Rule of thumb : Use inheritance *only if* doing so will remove `if` / `switch` statements from the caller code. (..) inheritance is not for code-reuse. You sometimes get a little code reuse via inheritance, but the primary purpose for inheritance is dynamic binding, and that is for flexibility. Composition is for code reuse, inheritance is for flexibility.

Bon ben j'aime encore moins C++...